

编译原理

编译原理

编译器是什么？

知识树

基本过程

词法分析

语言

正则语言

正则定义

如何让计算机识别用正则表达式定义的语言

NFA 非确定有限自动机

DFA 确定有限自动机

正则表达式转 NFA

直接用 NFA 识别语言

直接从正则表达式转 DFA

最小化 DFA 的算法

语法分析

语法的形式化：上下文无关文法

推导

推导 derivation

字符串符号文法

语法分析树

文法的二义性

文法二义性的消除

消除左递归

消除直接的左递归

消除间接的左递归

计算 first() 集合

计算 follow(A) 集合

LL(1) 文法的分析表

自底向上的文法分析

SLR 文法

LR(0) 项

扩充文法

自动机的过程

Closure of Item Sets

SLR 分析表的构建 (重点)

LR(1) 文法

构造 LR(1) 分析表

缺点

LALR 文法

语法制导定义

基本思想

举例

语法制导定义

继承属性

翻译模式

再次举例：中缀转后缀

扩展文法

扩展语法树

通过自顶向下的分析来实现先序遍历

实现先序遍历

Evaluation Order and Dependency Graphs

显式的语法分析树

S-属性制导定义

L-属性制导定义

需要满足三条规则：

语义分析和中间代码生成

Introduction

3 地址代码

1. 类型和声明

举例来说明翻译过程

2. 赋值和表达式

类型检查

3. 布尔表达式和流控制

流控制的语法制导定义

布尔表达式的语法制导定义

运行时环境

内存管理

stack 和活动记录

活动树

活动记录 (帧)

进程内通信

堆管理

多线程

垃圾回收

代码生成

指令选择

寄存器分配和赋值

指令调度

抽象目标状态机

指令集

基本块

流图

生存期和后续使用信息

简单代码生成器

代码优化

窥孔优化

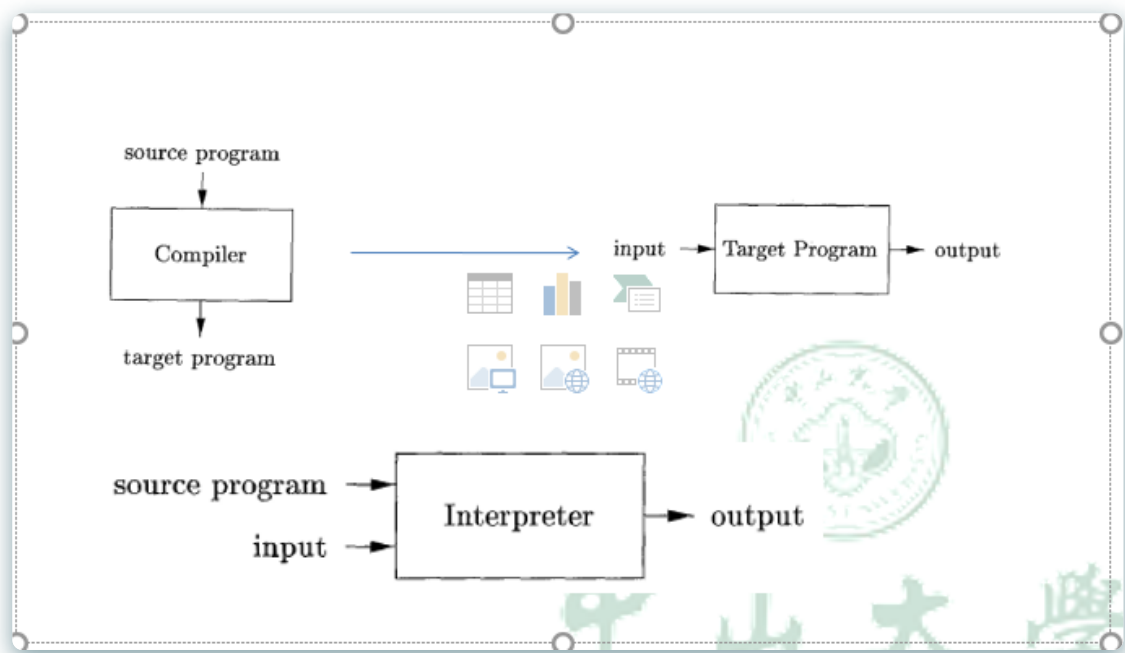
局部优化

控制流分析和循环优化

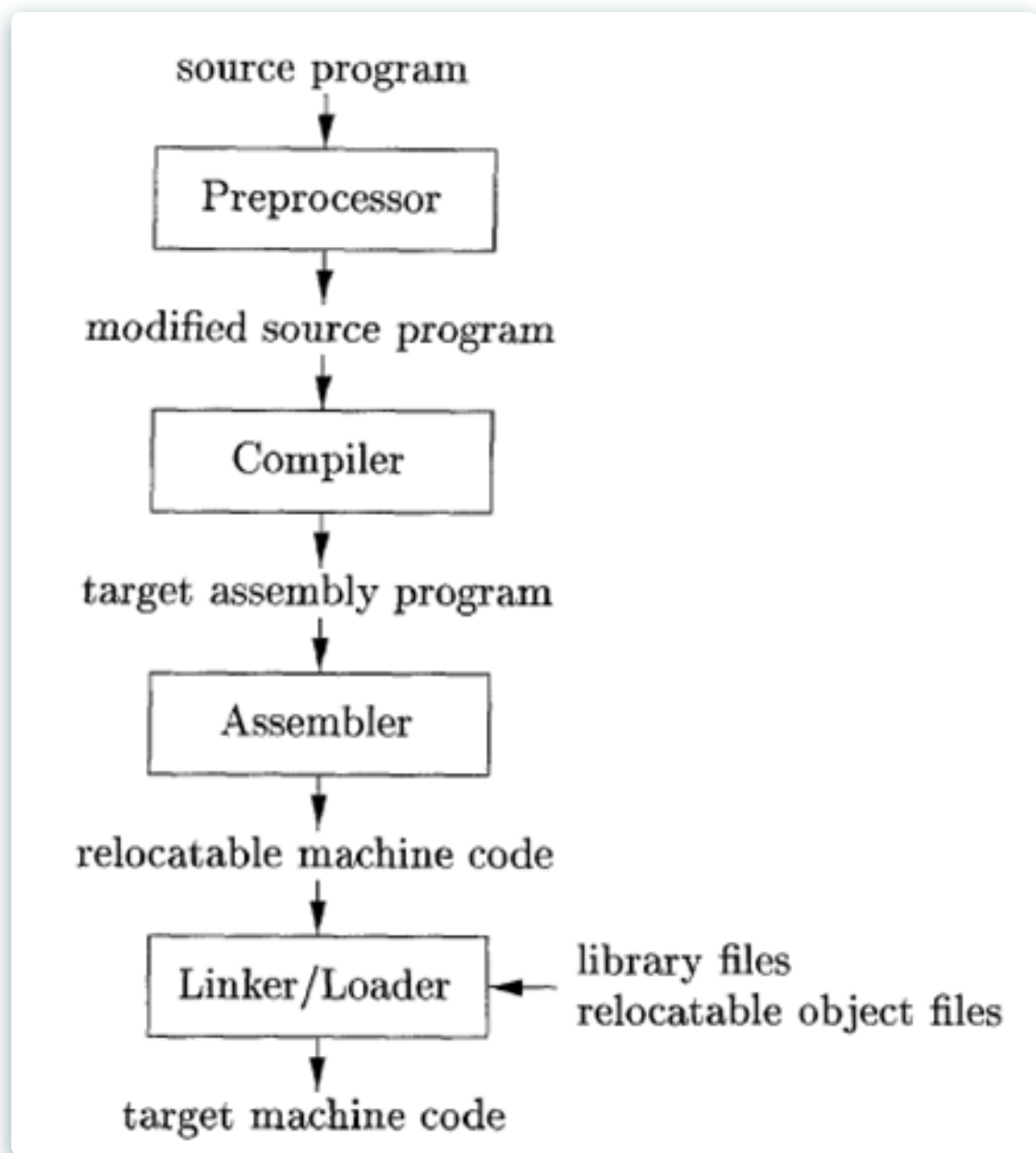
消除共同子表达式

✓ 编译器是什么？

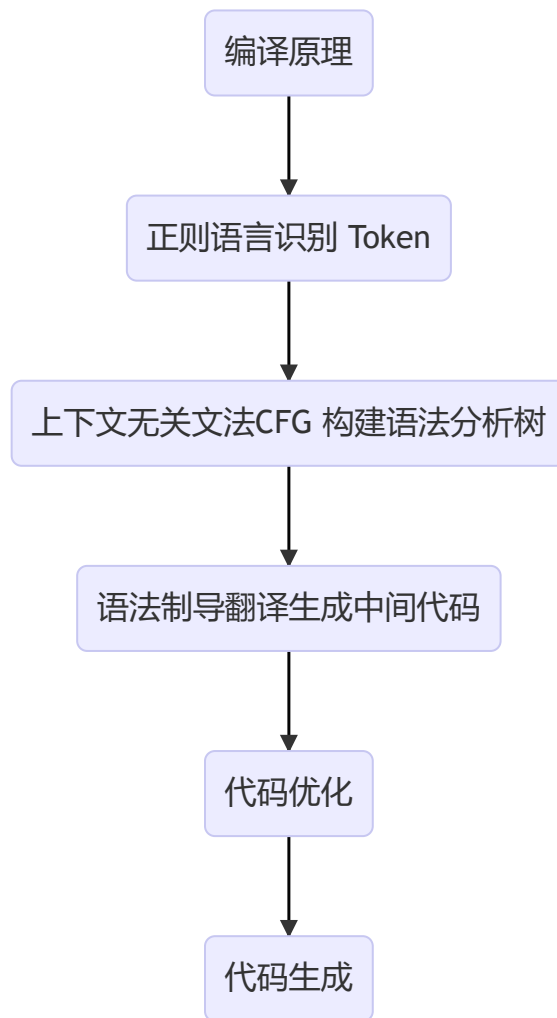
编译器是一个程序，主要是用来把源程序转换成另外一种计算机语言的程序。



语言编译的全过程：



✓ 知识树



编译原理是一种语言处理器，它完成了很多工作。

编译原理的过程->编译器->汇编器->链接器->加载器。

✓ 基本过程

过程是：

- 词法分析：读入源程序的字符流，组织成有意义的词素（lexeme）序列。或者称之为转换成单词（token）序列。
 - 词法分析器将其转化为token的序列。

- **token是指带有附加信息的字符串**

- 例如<“month”, id>
- 单纯的字符串我们称为lexeme

- **在自然语言中：名词，动词，形容词……**

- **程序语言中：标识符，关键字，数字……**

- **类别属性是包括在token中的基本信息**

- 任务2：给出单词的类别和一些相关属性值
- 属性值放在公用的符号表（Symbol Table）中，词法分析过程返回单词的属性值存放的地址
- 语法分析：建立语法分析树。
- 语义分析：使用语法树和符号表中的信息来检查源程序是否和**语言定义**的**语义**一致。
 - 类型检查
 - 类型转换
 - 其他
- 中间代码的生成：中间代码介于源语言和目标语言之间
 - 例如三地址代码
 - 好处：实现了前后端的分离
- 代码优化
- 代码生成

✓ 词法分析

词法分析器将其转换为 token 的序列，关键在于如何识别出 token。

- token 是指带有附加信息的字符串
- 每次获得一个token，对不同类别 token 的处理截然不同

在处理的时候，不能把编程语言当作成自然语言，那是因为自然语言的繁琐、不精确、难以别计算机 处理的特点，所以要**借助形式化的语言**！

语言

字母表：字符的集合

语言：字母表 Σ 上的语言，是由 Σ 中字符组成的字符串的集合

正则语言

正则语言的定义：

递归定义：

• **Basis:**

- ϵ is a regular expr; $L(\epsilon) = \{\epsilon\}$.
- $a \in \Sigma$ is a regular expr; $L(a) = \{a\}$.

• **Induction: if r and s are regular exprs,**

- $r \mid s$ is a regular expr; $L(r \mid s) = L(r) \cup L(s)$.
- rs is a regular expr; $L(rs) = L(r) L(s)$.
- r^* is a regular expr; $L(r^*) = (L(r))^*$.
- (r) is a regular expr; $L((r)) = L(r)$.

正则定义

$$\begin{array}{lll} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \dots & \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d 's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

无符号浮点数的正则表达式：

$$\begin{array}{lll} digit & \rightarrow & 0 \mid 1 \mid \dots \mid 9 \\ digits & \rightarrow & digit \, digit^* \\ optionalFraction & \rightarrow & . \, digits \mid \epsilon \\ optionalExponent & \rightarrow & (E (+ \mid - \mid \epsilon) \, digits) \mid \epsilon \\ number & \rightarrow & digits \, optionalFraction \, optionalExponent \end{array}$$

如何让计算机识别用正则表达式定义的语言

NFA 非确定有限自动机

Definition:

transition function: 给每个状态的每个字符都会生成 next states。

- 起始状态 s_0
- F : accept state

A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states S .
2. A set of input symbols Σ , the *input alphabet*. We assume that ϵ , which stands for the empty string, is never a member of Σ .
3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).
5. A set of states F , a subset of S , that is distinguished as the *accepting states* (or *final states*).

DFA 确定有限自动机

- 没有空迁移
- 对于每一个状态和每一个输入的字符，有且只有一个下一个状态

基于 DFA 的识别算法：

```
 $s = s_0;$   
 $c = nextChar();$   
while (  $c \neq eof$  ) {  
     $s = move(s, c);$   
     $c = nextChar();$   
}  
if (  $s$  is in  $F$  ) return "yes";  
else return "no";
```

时间复杂度： $O(|x|)$ ， x 为输入的字符串， $|x|$ 代表 x 的长度。

正则表达式转 NFA

这个写过实验相比应该很清楚了吧。

NFA 转 DFA 也很清楚了吧。

直接用 NFA 识别语言

```
1)   $S = \epsilon\text{-closure}(s_0);$ 
2)   $c = \text{nextChar}();$ 
3)  while (  $c \neq \text{eof}$  ) {
4)       $S = \epsilon\text{-closure}(\text{move}(S, c));$ 
5)       $c = \text{nextChar}();$ 
6)  }
7)  if (  $S \cap F \neq \emptyset$  ) return "yes";
8)  else return "no";
```

定理：对任何 DFA D 都存在一个正则表达式 r ，使得 $L(D)=L(r)$ 。

推论：对任何 NFA N 都存在一个正则表达式 r ，使得 $L(N)=L(r)$ 。

结论：DFA，NFA 和正则表达式三者的描述能力是一样的。

直接从正则表达式转 DFA

.....

最小化 DFA 的算法

1. 初始时，把所有状态分为两组： F ， $S-F$ ；
2. 然后不断的划分原 Group，对于每一个 Group 里面的两个状态，如果对里面的每一个状态对所有字符的迁移都是一样的 group 的话，那么它们就构成一个新的 group，并用新的 groups 代替原来的 group。
3. 直到 group 不变，算法结束。

4. 新的 DFA 的起始状态为包含了起始状态的状态，新的接受状态为包含了接受状态的集合。

多类 token 的识别：

- 取最长匹配串

留下的疑问：

- 如何构造正则表达式的语法树？
- 有些语言是正则语言无法表达的，怎么办？

第一问题的解答就是语法分析了，第二问的解答则是CFG。

✓ 语法分析

建立语法分析树。

语法分析的任务：

- 语法分析要解决的问题包括：
 - 从词法分析中获得的每个属性字 (token) 在语句中，或在整个程序中扮演什么角色
 - 例如：同样一个数字，在程序中可能是一个加数，可能是数组的下标，可能是循环的次数，可能是某个函数的参数等
 - 检查语句或程序是否符合程序语言的语法

解决问题之前，需要明确：

- 一个程序包括哪些语法成分？
- 这些语法成分以什么方式构成程序？

语法的形式化：上下文无关文法

一个上下文无关文法 (CFG) 包括四个部分

- 终端符号 (terminal) 的集合 T
- 非终端符号 (nonterminal) 的集合 N
- 唯一的开始符号 S ($S \in N$)
- 若干以下形式的产生式 (production):

$$X \rightarrow Y_1 Y_2 \dots Y_n, X \in N, Y_i \in T \cup N \cup \{\epsilon\}$$

一些方便的规定：

- 第一个产生式左端的非终端符号为开始符号
- 多个形如 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$ 的产生式可简写为 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$
- A,B,C 等表示非终端符号
- α, β, γ 表示由终端符号和非终端符号组成的串

推导

推导 derivation

从**开始符号**出发，每一步推导，用一个**产生式的右方**取代**左端的非终端符号**。使得整个文法只有终端符号。

字符串符号文法

我们将**由开始符号**推导出来的**只含有终端符号的串**称为这个**文法的句子**，一个**文法G定义的语言**就是这个文法**所有句子的集合**，记为 $L(G)$ 。判断一个终端符号串x是否**符合G的语法**就是判断**x是否属于 $L(G)$** 。

- 最左推导：每步推导都替换**最左边的非终端符号**
- 最右推导：每步推导都替换**最右边的非终端符号**

语法分析树

语法分析树是什么？

简单来说，就是从起始符号出发，把推导的字符从左往右作为**子结点**，其中这些**子结点的父节点**为它们推导前**被替换的节点**。

参考 CH4 note里面的例子

语法分析的过程实际上就是**构造分析树的过程**。

问题：一个句子可能有多个推导，那一个句子是否可能有多个分析树？答：可能。

文法的二义性

文法的二义性定义：如果对于一个文法，存在一个句子，对这个句子可以构造两颗不同的分析树，那么我们称这个文法为二义的。

文法二义性的消除

额.....没有通用的方法。

消除左递归

消除直接的左递归

对于任意一个具有如下形式的左递归：

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

可以替换成如下形式，从而消去左递归：

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

上面的消除方法也是很直观的，但是只能消除直接的左递归。

消除间接的左递归

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ -productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G . Note that the resulting non-left-recursive grammar may have ϵ -productions. \square

- 1) arrange the nonterminals in some order A_1, A_2, \dots, A_n .
- 2) **for** (each i from 1 to n) {
- 3) **for** (each j from 1 to $i - 1$) {
- 4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \cdots \mid \delta_k \gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j -productions
- 5) }
- 6) eliminate the immediate left recursion among the A_i -productions
- 7) }

注：Fig.4.11 指的就是上图。

上述算法也是比较直观的，把非终端符号进行排序，然后每次消除一个非终端符号的左递归。

计算 first() 集合

里面可以是终端符号也可以非终端符号，表示**第一个推导出来的非终端符号**，包含 ϵ 。

计算 follow(A) 集合

直观意思是 **A 后面可以接上 a 即 Aa，那么 a 在 follow(A)中**；

计算方式如下：

- $\$$ 在 S 的 follow 中，S 为起始非终端符号；
- 对于任意一个 $A \rightarrow \beta B \alpha$ ，那么 $first(\alpha)$ 中除了 ϵ 均在 follow(B) 中；
- 对于任意一个 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B \beta$ ，如果 $first(\beta)$ 包含 ϵ ，那么任何在 follow(A) 中的元素均在 follow(B) 中。

LL(1) 文法的分析表

分析表 M[A,a]: A 是非终端符号，a 是终端符号以及特殊字符 $\$$ (注： ϵ 不是终端符号 or 非终端符号)

- 当 A 遇到终端字符 a 或 $\$$ 时，对于所有 A 的如下形式的产生式： $A \rightarrow \alpha$ ：
 - 如果 a 在 $first(\alpha)$ 中，可以选所有 $A \rightarrow \alpha$ 的产生式
 - 如果 ϵ 在 $first(\alpha)$ 中，并且 a 在 follow(A) 中，那么可以选 $A \rightarrow \alpha$

一种**自顶向下**的文法分析。

自底向上的文法分析

LR Parsing, LR 分析。从左至右扫描，构造一个最右推导的逆过程。下面介绍几种具体的 LR 的实现。

SLR 文法

LR(0) 项

直观解释：就是多了一个点，在原有的推导项中，引入一个点，点表示位置，这样的项称为 LR(0) 项。

扩充文法

If G is a grammar with start symbol S, then G' , the augmented grammar for G, is G with a new start symbol S' and production $S' \rightarrow S$.

这样做的目的是，一旦归约到 S' ，那么这个过程就结束了。因为如果是 S，可能 S 只是中间的一个产生式。

自动机的过程

只有点在**非终端符号或终端符号的左边**，才继续要进行。

当有一个符号进来，会变到哪个状态。

Closure of Item Sets

项集合 I 的闭包包括：

- 它自身
- 对于在闭包中所有形如： $A \rightarrow \alpha.B\beta$ ，那么所有的 B 的推导： $B \rightarrow \cdot\alpha$ 也加入到闭包中。

SLR 分析表的构建（重点）

```
1 input: 扩增文法  $G'$ 
2 output: SLR分析表: ACTION 和 GOTO 表
3 METHOD:
4 1. 构造  $G'$  的规范集族  $C(LR(0)\text{自动机})$ 
5 2. 状态  $i$  是从  $I_i$  构造的. 那么状态  $i$  的动作函数被下面确定:
6   a. if  $[A \rightarrow \alpha.a\beta]$  in  $I_i$  and  $GOTO(I_i, a) = I_j$ , 这时候设为  $action(i, a) = s_j(\text{shift } j)$ 
7   b. if  $[A \rightarrow \alpha.]$  in  $I_i$ ,  $action(i, a) = r(A \rightarrow \alpha)(\text{reduce, 使用 } A \rightarrow \alpha \text{ 这条产生式}),$  其中  $a$  in  $FOLLOW(A)$  中
8   c. if  $[S' \rightarrow S.]$  in  $I_i$ , then  $action(i, \$) = acc$ 
```

LR(1) 文法

改进 LR(0) 文法，改进的地方在计算 closure 闭包和 goto 表时：

- 在计算一个 $[A \rightarrow \alpha.B\beta, a]$ 项时，含有的推导 $B \rightarrow \gamma$ 时，添加所有的项形如 $[B \rightarrow \cdot\gamma, b]$ ，其中只要任意 b 在 $\text{first}(\beta a)$ 中。
- 转移的时候，即遇到文法符号， $\text{goto}(I, X)$ 的计算过程：
 - 先把根据 X 而右移的那些推导加入进来，加入的时候保留后面的 $[a]$ ；
 - 然后再计算它们的闭包。

算法 4.53 LR(1)项集族的构造方法。

输入：一个增广文法 G' 。

输出：LR(1)项集族，其中的每个项集对文法 G' 的一个或多个可行前缀有效。

方法：过程 CLOSURE 和 GOTO，以及用于构造项集的主例程 items 见图 4-40。

```

SetOfItems CLOSURE( $I$ ) {
    repeat
        for (  $I$  中的每个项  $[A \rightarrow \alpha \cdot B\beta, a]$  )
            for (  $G'$  中的每个产生式  $B \rightarrow \gamma$  )
                for ( FIRST( $\beta a$ ) 中的每个终结符号  $b$  )
                    将  $[B \rightarrow \cdot \gamma, b]$  加入到集合  $I$  中;
    until 不能向  $I$  中加入更多的项;
    return  $I$ ;
}

SetOfItems GOTO( $I, X$ ) {
    将  $J$  初始化为空集;
    for (  $I$  中的每个项  $[A \rightarrow \alpha \cdot X\beta, a]$  )
        将项  $[A \rightarrow \alpha X \cdot \beta, a]$  加入到集合  $J$  中;
    return CLOSURE( $J$ );
}

void items( $G'$ ) {
    将  $C$  初始化为 {CLOSURE} ({ $[S' \rightarrow \cdot S, \$]$ });
    repeat
        for (  $C$  中的每个项集  $I$  )
            for ( 每个文法符号  $X$  )
                if ( GOTO( $I, X$ ) 非空且不在  $C$  中 )
                    将 GOTO( $I, X$ ) 加入  $C$  中;
    until 不再有新的项集加入到  $C$  中;
}

```

图 4-40 为文法 G' 构造 LR(1)项集族的算法

```

SetOfItems CLOSURE( $I$ ) {
    repeat
        for ( each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in  $I$  )
            for ( each production  $B \rightarrow \gamma$  in  $G'$  )
                for ( each terminal  $b$  in FIRST( $\beta a$ ) )
                    add  $[B \rightarrow \cdot \gamma, b]$  to set  $I$ ;
    until no more items are added to  $I$ ;
    return  $I$ ;
}

```

```

SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha \cdot X\beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}

```


构造 LR(1) 分析表

算法 4.56 规范 LR 语法分析表的构造。

输入：一个增广文法 G' 。

输出： G' 的规范 LR 语法分析表的函数 ACTION 和 GOTO。

方法：

1) 构造 G' 的 LR(1) 项集族 $C' = \{I_0, I_1, \dots, I_n\}$ 。

2) 语法分析器的状态 i 根据 I_i 构造得到。状态 i 的语法分析动作按照下面的规则确定：

① 如果 $[A \rightarrow \alpha \cdot a\beta, b]$ 在 I_i 中，并且 $\text{GOTO}(I_i, a) = I_j$ ，那么将 $\text{ACTION}[i, a]$ 设置为“移入 j ”。这里 a 必须是一个终结符号。

② 如果 $[A \rightarrow \alpha \cdot, a]$ 在 I_i 中且 $A \neq S'$ ，那么将 $\text{ACTION}[i, a]$ 设置为“规约 $A \rightarrow \alpha$ ”。

③ 如果 $[S' \rightarrow S \cdot, \$]$ 在 I_i 中，那么将 $\text{ACTION}[i, \$]$ 设置为“接受”。

如果根据上述规则会产生任何冲突动作，我们就说这个文法不是 LR(1) 的。在这种情况下，这个算法无法为该文法生成一个语法分析器。

3) 状态 i 相对于各个非终结符号 A 的 goto 转换按照下面的规则构造得到：如果 $\text{GOTO}(I_i, A) = I_j$ ，那么 $\text{GOTO}[i, A] = j$ 。

4) 所有没有按照规则(2)和(3)定义的分析表条目都设为“报错”。

5) 语法分析器的初始状态是由包含 $[S' \rightarrow \cdot S, \$]$ 的项集构造得到的状态。 \square

由算法 4.56 生成的语法分析动作和 GOTO 函数组成的表称为规范 LR(1) 语法分析表。使用这个表的 LR 语法分析器称为规范 LR(1) 语法分析器。如果语法分析动作函数中不包含多重定义的条

构造 action 表时，

- 做移进操作时，和之前 LR(0) 项一致不变。
- 作归约时， $[A \rightarrow \alpha \cdot, a]$ ，设置 $\text{action}(i, a)$ 为归约，而不是使用 follow 集合。
- acc 也是类似。

如果得到的分析动作函数中不包含多重定义的条目，那么给定文法就成为 LR(1) 文法。

缺点

- 状态数太多，是 LR(0) 文法状态数的几倍以上！

LALR 文法

lookahead LR parsing. 使用范围最广的就是 LALR 文法。

主要思想就是合并 LR(1) 的自动机的某些状态。具体合并即使相同的 LR(0) 项即合并。

✓ 语法制导定义

制导定义翻译是什么？

- Translation Process guided by Context-Free Grammars

语法制导定义的应用

- 覆盖了编译的两个阶段

- 语义分析
- 中间代码生成

基本思想

- 给每个文法符号关联上属性
 - 根据具体的应用
 - data 可以被处理
- 把产生式关联上一些语义
 - 也被称为语义动作
 - 操作属性

举例

语法制导定义

见如下的表：产生式 + 语义规则

举例：表达式求值

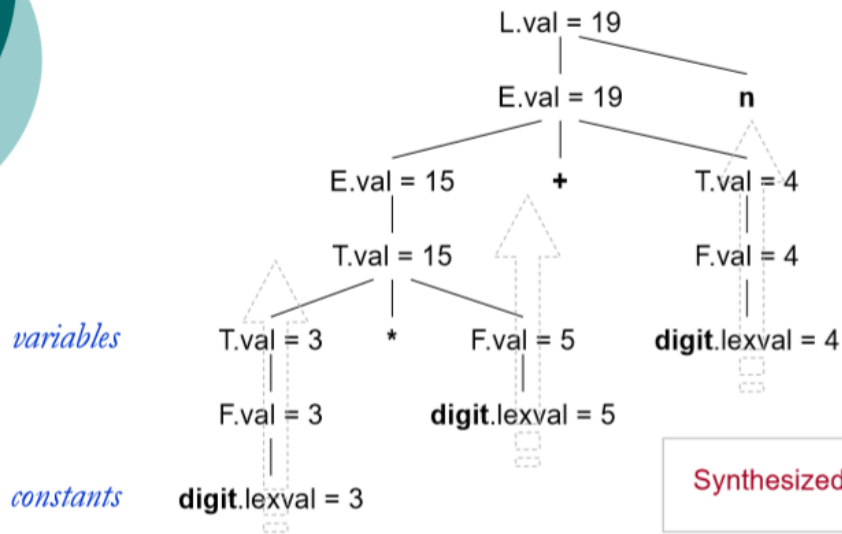
Syntax-Directed Definition

	Productions	Semantic Rules
1	$L \rightarrow E n$	$L.val = E.val$ $print(L.val)$ Side-effects
2	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3	$E \rightarrow T$	$E.val = T.val$
4	$T_1 \rightarrow T_2 * F$	$T_1.val = T_2.val * F.val$
5	$T \rightarrow F$	$T.val = F.val$
6	$F \rightarrow (E)$	$F.val = E.val$
7	$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

Different
subfix style

语法分析树

Annotated Parse Tree



继承属性

再次举例：有理数的表示

Inherited Attribute

No.	Productions	Semantic Rules
1	$D \rightarrow T L$	$L.inh = T.type$
2	$T \rightarrow \mathbf{int}$	$T.type = \mathbf{INTEGER}$
3	$T \rightarrow \mathbf{real}$	$T.type = \mathbf{REAL}$
4	$L \rightarrow L_1, \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

翻译模式

再次举例：中缀转后缀

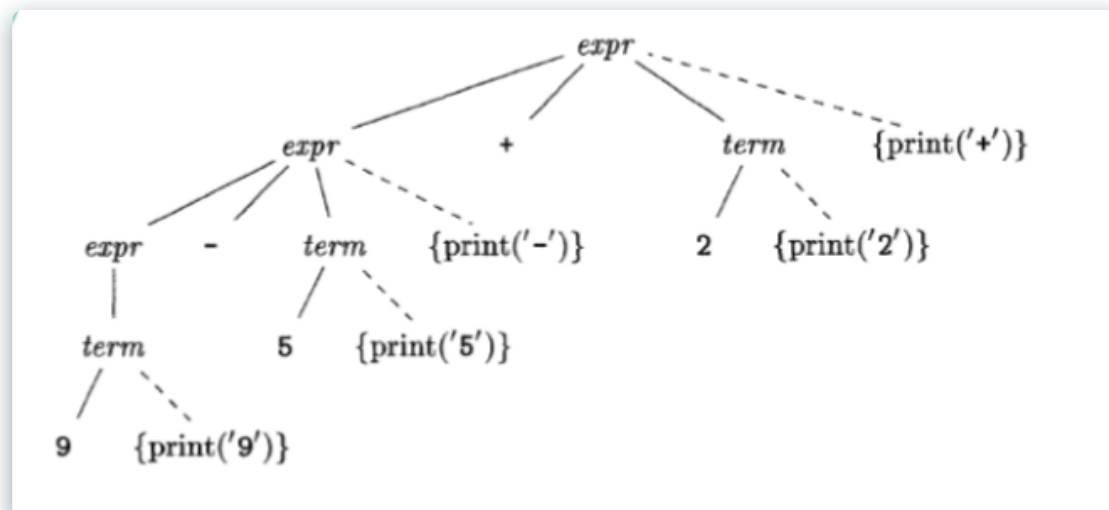
扩展文法

$expr$	\rightarrow	$expr_1 + term$	$\{print('+')\}$
$expr$	\rightarrow	$expr_1 - term$	$\{print('-')\}$
$expr$	\rightarrow	$term$	
$term$	\rightarrow	0	$\{print('0')\}$
$term$	\rightarrow	1	$\{print('1')\}$
		...	
$term$	\rightarrow	9	$\{print('9')\}$

大括号中的语句称为action，这一系列产生式称为Translation Scheme

扩展语法树

如果将 action 视作产生式右端的一部分，则对于上述句子可以得到扩展了的语法树。



对于上述语法分析树进行先序遍历，则由各个 action 的语句和执行顺序，可以得到后缀表达式。

通过自顶向下的分析来实现先序遍历

这里自顶向下即从起始符号出发，不停的 lookahead，然后选取特定的产生式。

将上述文法进行转换，变为：

$$\begin{array}{lll}
 \text{expr} & \rightarrow & \text{term rest} \\
 \text{rest} & \rightarrow & + \text{term} \{ \text{print}(' + ') \} \text{rest} \\
 & | & - \text{term} \{ \text{print}(' - ') \} \text{rest} \\
 & | & \epsilon \\
 \text{term} & \rightarrow & 0 \{ \text{print}(' 0 ') \} \\
 & | & 1 \{ \text{print}(' 1 ') \} \\
 & & \dots \\
 & | & 9 \{ \text{print}(' 9 ') \}
 \end{array}$$

实现先序遍历

将action 中的内容照搬到程序的恰当位置中，生成下列程序：

```

void expr() {
    term(); rest();
}

void rest() {
    if ( lookahead == '+' ) {
        match('+'); term(); print('+'); rest();
    }
    else if ( lookahead == '-' ) {
        match('-'); term(); print('-'); rest();
    }
    else { } /* do nothing with the input */ ;
}

void term() {
    if ( lookahead is a digit ) {
        t = lookahead; match(lookahead); print(t);
    }
    else report("syntax error");
}

```

编译原理书上消解加减乘除的文法的过程：

如果 $*$ 先于 $+$ 获得运算分量, 我们就说 $*$ 比 $+$ 具有更高的优先级。在通常的算术中, 乘法和除法比加法和减法具有更高的优先级。因此在表达式 $9 + 5 * 2$ 和 $9 * 5 + 2$ 中, 都是运算分量 5 首先参与 $*$ 运算, 即这两个表达式分别等价于 $9 + (5 * 2)$ 和 $(9 * 5) + 2$ 。

例 2.6 算术表达式的文法可以根据表示运算符结合性和优先级的表格来构建。我们首先考虑四个常用的算术运算符和一个优先级表。在此优先级表中, 运算符按照优先级递增的顺序排列, 同一行上的运算符具有相同的结合性和优先级:

左结合: $+$ $-$

左结合: $*$ $/$

我们创建两个非终结符号 $expr$ 和 $term$, 分别对应于这两个优先级层次, 并使用另一个非终结符号 $factor$ 来生成表达式中的基本单元。当前, 表达式的基本单元是数位和带括号的表达式。

$factor \rightarrow \text{digit} \mid (expr)$

现在我们考虑具有最高优先级的二目运算符 $*$ 和 $/$ 。由于这些运算符是左结合的, 因此其产生式和左结合列表的产生式类似:

$term \rightarrow term * factor$
 $\mid term / factor$
 $\mid factor$

类似地, $expr$ 生成由加减运算符分隔的 $term$ 列表:

$expr \rightarrow expr + term$
 $\mid expr - term$
 $\mid term$

因此最终得到的文法是:

$expr \rightarrow expr + term \mid expr - term \mid term$
 $term \rightarrow term * factor \mid term / factor \mid factor$
 $factor \rightarrow \text{digit} \mid (expr)$

Evaluation Order and Dependency Graphs

一般框架的步骤:

- 把属性关联上文法符号
- 给每条产生式定义语义规则
- 根据语法分析树画出依赖图
- 根据图的拓扑排序结构确定计算顺序
- 根据计算顺序执行语义规则

效率太低, 不采用。

所以在生成语法分析树的时候就计算好属性值。

显式的语法分析树

- S-属性, 综合属性
 - 如果某个实现在语法分析树上的值, 是由该节点的子结点以及它本身的属性值确定的;
- L-属性, 继承属性
 - 如果该节点的值是由父亲或左兄弟继承而来。

相应的定义:

- S-属性制导定义
 - 所有属性都是综合属性
- L-属性制导定义
 - 每个属性都是综合属性或继承属性

S-属性制导定义

计算的顺序是自底向上，和 LR 文法解析的顺序一致。

举例：

The Previous Calculator Example

No.	Productions	Semantic Rules
1	$L \rightarrow E n$	$L.val = E.val$ $print(L.val)$
2	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3	$E \rightarrow T$	$E.val = T.val$
4	$T_1 \rightarrow T_2 * F$	$T_1.val = T_2.val * F.val$
5	$T \rightarrow F$	$T.val = F.val$
6	$F \rightarrow (E)$	$F.val = E.val$
7	$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

Associate attributes of grammar symbols with positions in the parsing stack.

LR 解释的实现：

Implementation in LR Parsing

No.	Productions	Code	Notes
1	$L \rightarrow E n$	stack[ntop].val = stack[top - 1].val; print(stack[ntop].val);	
2	$E \rightarrow E_1 + T$	stack[ntop].val = stack[top - 2].val + stack[top].val;	stack[top - 1].val = '+'
3	$E \rightarrow T$		
4	$T_1 \rightarrow T_2 * F$	stack[ntop].val = stack[top - 2].val * stack[top].val;	stack[top - 1].val = '*'
5	$T \rightarrow F$		
6	$F \rightarrow (E)$	stack[ntop].val = stack[top - 1].val;	stack[top].val = ')' stack[top - 2].val = '('
7	$F \rightarrow \text{digit}$		

Setup an attribute stack with the same height as parsing (state) stack:
 $\text{ntop} = \text{top} - | \text{right-side} | + 1$

Evaluation While LR Parsing

Step	States (Illustrative)	Attributes	Input	Code	Output
1	\$	\$	3 * 5 + 4 n \$		
2	\$ 3	\$ 3	* 5 + 4 n \$	--	F → digit
3	\$ F	\$ 3	* 5 + 4 n \$	--	T → F
4	\$ T	\$ 3	* 5 + 4 n \$		
5	\$ T *	\$ 3 *	5 + 4 n \$		
6	\$ T * 5	\$ 3 * 5	+ 4 n \$	--	F → digit
7	\$ T * F	\$ 3 * 5	+ 4 n \$	3 * 5	T → T * F
8	\$ T	\$ 15	+ 4 n \$	--	E → T
9	\$ E	\$ 15	+ 4 n \$		
10	\$ E +	\$ 15 +	4 n \$		
11	\$ E + 4	\$ 15 + 4	n \$	--	F → digit
12	\$ E + F	\$ 15 + 4	n \$	--	T → F
13	\$ E + T	\$ 15 + 4	n \$	15 + 4	E → E + T
14	\$ E	\$ 19	n \$		
15	\$ E n	\$ 19 n	\$	print(19)	L → E n
16	\$ L	\$ 19	\$	accept	

L-属性制导定义

L-属性的定义：

- 不能依赖任何是综合属性的父母
- 该翻译模式和 depth-first 先序遍历语法分析树一致；
 - 从左至右
 - 自上而下

- 是一个自顶向下的语法分析过程

需要满足三条规则：

- A 的继承属性必须在 A 之前计算好
- 一个 action 必须不和他右边的综合属性产生依赖
- 一个综合属性的计算必须在与他关联的属性都被计算完之后再计算。

✓ 语义分析和中间代码生成

Introduction

IR: intermediate representation 中间表示

语义分析 (Semantic analysis) (static checking)

- **类型检查**
- 未使用的代码检查
- 未使用的变量检查

高层次的中间表示：

- 语法树和 DAG
 - 适合做静态类型检查

低层次中间代码表示

- 3 地址代码
 - 适合给机器依赖的任务，比如寄存器分配和指令选择

3 地址代码

形式：

- 一条指令的右侧 (赋值运算符右边) 最多只有一个运算符

基本概念：

- 地址和指令

地址可以具有如下形式之一：

- 名字。允许源程序的名字作为三地址代码中的地址。
- 常量
- 编译器生成的临时变量

常见的三地址指令形式：

- 形如 $x = y \text{ op } z$
- $x = \text{op } y$
- $x = y$
- `goto L`
- `if x goto L` 或 `if False x goto L`
- `if x relop y goto L`; `relop` 为 `<`, `==`
- 过程调用和返回通过下列指令来实现：
 - `param x` 进行参数传递
 - `call p, n`
- $x = y[i]$ 和 $x[i] = y$
- $x = \&y$ 、 $x = *y$ 、 $*x = y$

下面将介绍如下形式的语义分析和语法制导定义：

- 类型和声明
- 赋值和表达式
- 类型检查
- 布尔表达式和流控制

1. 类型和声明

声明

- 变量：显式的
- 其他名字：显式的

在强语言类型中会进行类型检查

- 类型完整性
- 类型推断
- 隐式类型转换
- 解析重载运算符

举例来说明翻译过程

○ Declare only one name at a time

$D \rightarrow T \text{ id} ; D \mid \varepsilon$
 $T \rightarrow B C \mid \text{record} \{ D \}$
 $B \rightarrow \text{int} \mid \text{double}$
 $C \rightarrow [\text{num}] C \mid \varepsilon$

- 引入语法制导定义

○ Computing types and their widths

$T \rightarrow B$ $\{ t = B.type; w = B.width \}$
 C $\{ T.type = C.type; T.width = C.width \}$
 $B \rightarrow \text{int}$ $\{ B.type = \text{INTEGER}; B.width = 4 \}$
 $B \rightarrow \text{double}$ $\{ B.type = \text{DOUBLE}; B.width = 8 \}$
 $C \rightarrow [\text{num}] C_1$ $\{ C.type = \text{array}(\text{num.value}, C_1.type);$
 $C.width = \text{num.value} \times C_1.width \}$
 $C \rightarrow \varepsilon$ $\{ C.type = t; C.width = w \}$

Just try it: `int[2][3]`
What is `T.type` and `T.width` ?

Type
expression

○ Computing relative addresses

$P \rightarrow$ $\{ \text{offset} = 0 \}$

D

$D \rightarrow T \text{ id} ;$ $\{ \text{top.put}(\text{id.lexeme}, T.type, \text{offset});$
 $\text{offset} += T.width \}$

D_1

$D \rightarrow \varepsilon$

top denotes the
current symbol table

○ Field names in records

```

T → record    { Env.push(top); top = new Env();
                Stack.push(offset); offset = 0; }

D end          { T.type = record(top); T.width = offset;
                top = Env.pop(); offset = Stack.pop(); }

```

2. 赋值和表达式

中间代码生成

- 代码连接
- 增量生成

语法制导定义如下：

○ Code concatenation (syntax-directed definition)

	Productions	Semantic Rules
1	$S \rightarrow \text{id} = E ;$	$S.\text{code} = E.\text{code} \parallel \text{gen}(\text{top.get}(\text{id.lexeme}) \text{'=' } E.\text{addr})$
2	$E \rightarrow E_1 + E_2$	$E.\text{addr} = \text{new Temp}();$ $E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel \text{gen}(E.\text{addr} \text{'=' } E_1.\text{addr} \text{'+' } E_2.\text{addr})$
3	$E \rightarrow - E_1$	$E.\text{addr} = \text{new Temp}();$ $E.\text{code} = E_1.\text{code} \parallel \text{gen}(E.\text{addr} \text{'=' 'minus' } E_1.\text{addr})$
4	$E \rightarrow (E_1)$	$E.\text{addr} = E_1.\text{addr};$ $E.\text{code} = E_1.\text{code}$
5	$E \rightarrow \text{id}$	$E.\text{addr} = \text{top.get}(\text{id.lexeme});$ $E.\text{code} = ''$

C++中的声明的翻译模式

○ Translation scheme

```
S → id = E ;    { gen(top.get(id.lexeme) '=' E.addr) }
S → L = E ;     { gen(L.array.base '[' L.addr ']' '=' E.addr) }
E → E1 + E2   { E.addr = new Temp();
                  gen(E.addr '=' E1.addr '+' E2.addr) }

E → id          { E.addr = top.get(id.lexeme) }
E → L           { E.addr = new Temp();
                  gen(E.addr '=' L.array.base '[' L.addr ']') }

L → id [ E ]    { L.array = top.get(id.lexeme);
                  L.type = L.array.type.element;
                  L.addr = new Temp();
                  gen(L.addr '=' E.addr '*' L.type.width) }

L → L1 [ E ]    { L.array = L1.array;
                  L.type = L1.type.element;
                  t = new Temp();
                  L.addr = new Temp();
                  gen(t '=' E.addr '*' L.type.width);
                  gen(L.addr '=' L1.addr '+' t) }
```

类型检查

强类型语言会进行类型检查。

○ Type checking, inference and implicit casting

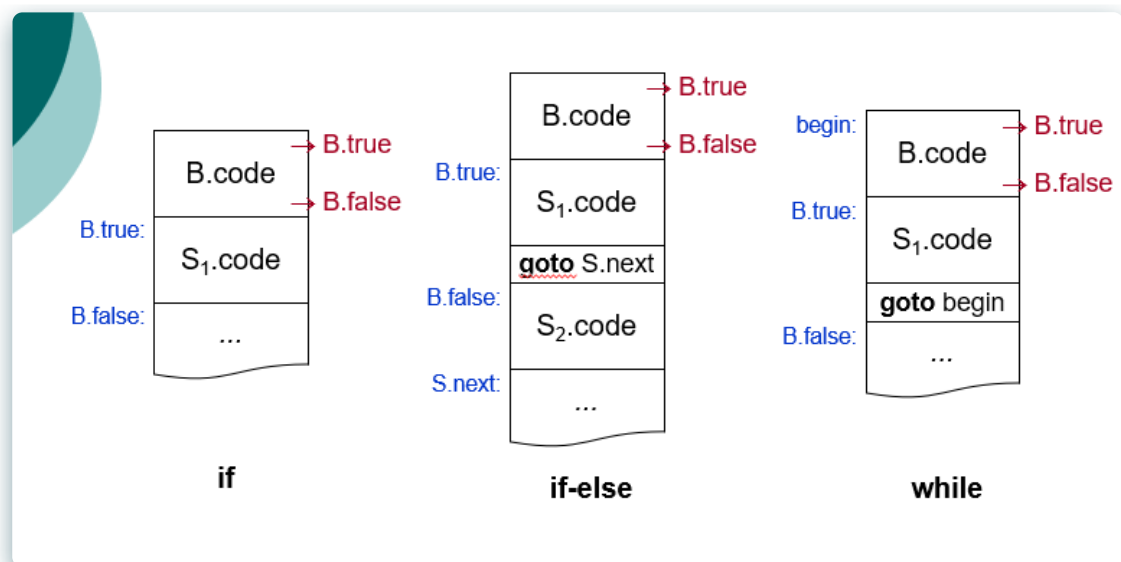
```
E → E1 * E2 { E.addr := new Temp();
                  if (E1.type == TK_INT && E2.type == TK_INT) {
                      gen(E.addr '=' E1.addr '*int' E2.addr);
                      E.type = TK_INT;
                  }
                  elseif (E1.type == TK_REAL && E2.type == TK_REAL) {
                      gen(E.addr '=' E1.addr '*real' E2.addr);
                      E.type = TK_REAL;
                  }
                  elseif (E1.type == TK_INT && E2.type == TK_REAL) {
                      t := new Temp();
                      gen(t '=' 'int2real' E1.addr);
                      gen(E.addr '=' t '*real' E2.addr);
                      E.type = TK_REAL;
                  }
                  elseif (...) { ... }
              }
```

3. 布尔表达式和流控制

流控制的声明：

- S -> **if** (B) S1
- S -> **if** (B) S1 **else** S2
- S -> **while** (B) S1

生成代码的示意图：



流控制的语法制导定义

Productions	Semantic Rules
$P \rightarrow S$	$S.next = \text{new Label}();$ $P.code = S.code \parallel \text{label}(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign.code}$
$S \rightarrow S_1 S_2$	$S_1.next = \text{new Label}();$ $S_2.next = S.next;$ $S.code = S_1.code \parallel \text{label}(S_1.next) \parallel S_2.code$
$S \rightarrow \text{if (B) } S_1$	$B.true = \text{new Label}();$ $B.false = S_1.next = S.next;$ $S.code = B.code \parallel \text{label}(B.true) \parallel S_1.code$
$S \rightarrow \text{if (B) } S_1 \text{ else } S_2$	$B.true = \text{new Label}();$ $B.false = \text{new Label}();$ $S_1.next = S_2.next = S.next;$ $S.code = B.code \parallel \text{label}(B.true) \parallel S_1.code \parallel \text{gen('goto' S.next)} \parallel \text{label}(B.false) \parallel S_2.code$
$S \rightarrow \text{while (B) } S_1$	$begin = \text{new Label}();$ $B.true = \text{new Label}();$ $B.false = S.next;$ $S_1.next = begin;$ $S.code = \text{label}(begin) \parallel B.code \parallel \text{label}(B.true) \parallel S_1.code \parallel \text{gen('goto' begin)}$

Where does `S.next` come from ?

Avoid redundant **gotos**

布尔表达式的语法制导定义

Syntax-Directed Definition for Booleans

Productions	Semantic Rules
$B \rightarrow B_1 \ \ B_2$	$B_1.true = B.true;$ $B_1.false = \text{new Label}();$ $B_2.true = B.true;$ $B_2.false = B.false;$ $B.code = B_1.code \ \ \text{label}(B_1.false) \ \ B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = \text{new Label}();$ $B_1.false = B.false;$ $B_2.true = B.true;$ $B_2.false = B.false;$ $B.code = B_1.code \ \ \text{label}(B_1.true) \ \ B_2.code$
$B \rightarrow ! \ B_1$	$B_1.true = B.false;$ $B_1.false = B.true;$ $B.code = B_1.code$
$B \rightarrow E_1 \ \text{relop} \ E_2$	$B.code = E_1.code \ \ E_2.code$ $\ \ \text{gen}('if' \ E_1.addr \ \text{relop.op} \ E_2.addr \ 'goto' \ B.true)$ $\ \ \text{gen}('goto' \ B.false)$
$B \rightarrow \text{true}$	$B.code = \text{gen}('goto' \ B.true)$
$B \rightarrow \text{false}$	$B.code = \text{gen}('goto' \ B.false)$

Short-Circuit
Evaluation

✓ 运行时环境

内存管理

典型的内存管理的布局：自上而下地址逐渐升高

- 代码区
- 静态区
- 堆
- free Memory
- stack 栈

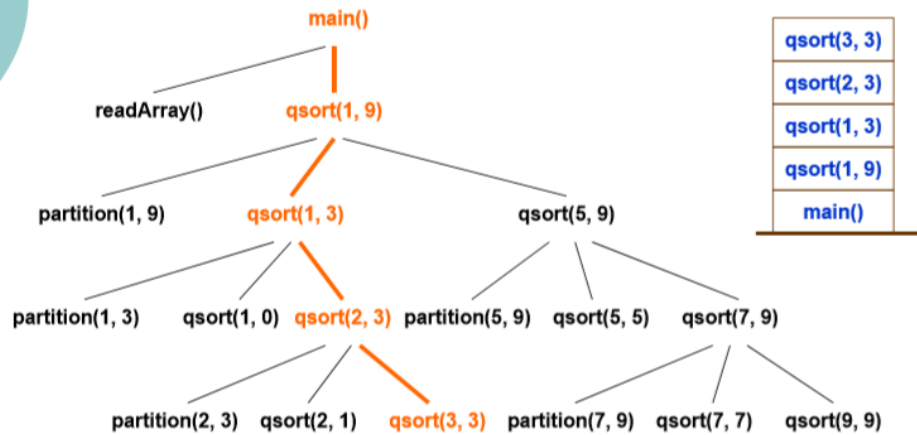
stack 和活动记录

- 实现子程序抽象
 - 也被称作控制栈
- 一般设计原则：
 - 按需激活

活动树

- 活动树和运行时的栈

○ Activation tree and run-time stack



活动记录（帧）

○ Typical organization of activation records

Actual Parameters	set by Caller, register preferred
Returned Values	set by Callee, register preferred
Control Link	pointer to Caller's AR
Access Link	pointer to the outer AR
Saved Machine Status	return address & registers
Local Data	user defined
Temporaries	compiler generated

每次调用的时候，会把一个活动记录存入栈。这应该是一个逻辑结构？

- Control Link: 指向 Caller's AR (活动记录)
- Access Link: 指向外部AR (活动记录)，比如在函数里面定义函数。所以需要指向该函数里面定义的私有函数（此时这个私有函数貌似存在外部？）
- Saved Machine Status: 地址，回到的地址；以及原来那个函数caller的寄存器变量存起来
- Local Data: 局部变量
- 临时变量：编译器生成的变量

进程内通信

实现调用过程。Caller 和 Callee 的通信。

- Calling sequence 调用步骤
 - 分配一个活动记录
 - 把相关信息写入相应的 AR 中
- 返回步骤
 - 恢复调用前的状态
 - 继续执行 Caller 的程序

堆管理

关键要解决的问题：内存碎片化。

- 时间效率和空间效率

如何最小化碎片？

多线程

多线程共享同一个堆，每个线程有它的控制栈；

多线程之间的通信也更加简单。

垃圾回收

嘿嘿，不考

✓ 代码生成

代码生成关注

- 指令选择
 - 选择合适的状态机器指令去实现 IR 的状态
- 寄存器分配和赋值
 - 利用所有的寄存器，快速计算
- 指令顺序

- 决定指令执行的调度顺序
- 也称之为指令调度

指令选择

需要考虑速度和花费的代码长度。

比如：

```
1 | LD R0, #0
2 | //
3 | XOR R0, R0
```

寄存器分配和赋值

包括两个子问题

- 寄存器分配
 - 选择将驻留在程序各点寄存器中的变量集
- 寄存器赋值
 - 选择变量将驻留的特定寄存器。

这是比较困难的一步。

指令调度

什么流水线啊，啥啥的，参考祭祖。

抽象目标状态机

指令集

基本块

每个基本块是满足下列的最大的连续三地址指令序列

- 控制流只能从基本块中的**第一个指令进入**该块。也就是说，没有跳转到基本块中间的转移指令
- **除了**基本块的**最后一个指令**，控制流在离开基本块之前**不会停机**或者**跳转**

即单入口和单退出。

构造基本块的算法：

算法 8.5 把三地址指令序列划分成为基本块。

输入：一个三地址指令序列。

输出：输入序列对应的一个基本块列表，其中每个指令恰好被分配给一个基本块。

方法：首先，我们确定中间代码序列中哪些指令是首指令 (leader)，即某个基本块的第一个指令。跟在中间程序末端之后的指令的不包含在首指令集合中。选择首指令的规则如下：

- 1) 中间代码的第一个三地址指令是一个首指令。
- 2) 任意一个条件或无条件转移指令的目标指令是一个首指令。
- 3) 紧跟在一个条件或无条件转移指令之后的指令是一个首指令。

然后，每个首指令对应的基本块包括了从它自己开始，直到下一个首指令 (不含) 或者中间程序的结尾指令之间的所有指令。 □

流图

流图的构造方式：

- 找出基本块
- 按照跳转规则标上有向边，最后添加 exit


生存期和后续使用信息

使用后向扫描算法。

- 初始化：对每一个变量 v
 - $v.\text{nextUse} = \text{none}$;
 - $v.\text{liveness} = v \text{ is temporary} ? \text{false} : \text{true}$;
- 对每一个 $i : x = y + z$
 - $x.\text{liveness} = \text{false}$;
 - $x.\text{nextUse} = \text{none}$;
 - $y.\text{liveness} = z.\text{liveness} = \text{true}$;
 - $y.\text{nextUse} = z.\text{nextUse} = i$;
- 这些信息将存在符号表里

An Example

- Information are stored at the entry of each variables in the symbol table

(1) $t = a - b$		(1) $t^{(3),T} = a^{(2),T} - b^{-,T}$
(2) $u = a - c$		(2) $u^{(3),T} = a^{-,T} - c^{-,T}$
(3) $v = t + u$		(3) $v^{(4),T} = t^{-,F} + u^{(4),T}$
(4) $d = v + u$		(4) $d^{-,T} = v^{-,F} + u^{-,F}$

Var.	Next-Use					Liveness				
	Init	(4)	(3)	(2)	(1)	Init	(4)	(3)	(2)	(1)
a	-			(2)	(1)	T			T	T
b	-				(1)	T				T
c	-			(2)		T			T	
d	-	-				T	F			
t	-		(3)		-	F		T		F
u	-	(4)	(3)	-		F	T	T	F	
v	-	(4)	-			F	T	F		

简单代码生成器

- simple:
 - 给每个基本块生成代码
- Motivation
 - 最大限度地充分利用寄存器

✓ 代码优化

窥孔优化

对局部代码非常有效的优化，比如一个基本块的优化

- 消除冗余的 Loads 和 Stores
- 消除不可到达的代码
- 流控制优化
- 代数简化和长度降低

○ Examples

- $x = x + 0$
// eliminated
- $x = x * 1$
// eliminated
- $y = x * 2$
 $y = x << 1$
- $y = x * 4$
 $y = x << 2$

局部优化

保留语义的转换

Transformations

- 共同子表达式
- 常量和复制传播
- 消除冗余操作

Basic Block => DAG => Basic Block

控制流分析和循环优化

不只是针对一个基本块优化。

涉及三个子问题：

- 如何基于一个流图定义一个循环
- 如何在流图中找到循环
- 如何优化循环

消除共同子表达式

- 比如，同时出现计算表达式（即赋值符号右边计算是一样的话，直接赋值）
- 与循环无关的计算，移动至循环外进行计算